TICAM Report 99-29

3D *hp*-ADAPTIVE FINITE ELEMENT PACKAGE FORTRAN 90 IMPLEMENTATION (3Dhp90)

L. Demkowicz, A. Bajer, W. Rachowicz, K. Gerdes

Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, TX 78712

Abstract

This document provides a description of the first release of a package supporting 3D Finite Element *hp*-approximations for the solution of various boundary-value problems. At the moment the discretization is defined only on unstructured hexahedral grids. The package supports both *h*- and *p*-refinements of the mesh.

Acknowledgment

The work has been supported by Air Force under Contract F49620-98-1-0255. The computations reported in this work were done through the National Science Foundation's National Partnership for Advanced Computational Infrastructure.

Contents

1 Introduction

This is a continuation of the work reported in [4, 5], in context of 3D discretizations. We recall the main differences between the present implementation and the earlier versions [1, 3]:

1. Data structure has been reduced to just two arrays. These arrays consist of user-defined objects called *elements* and *nodes*. Each of the objects has several attributes. There are two important advantages of this modification, made possible by FORTRAN 90:
   - the actual code is more readible (all names are self-explanatory),
   - a parallel, distributed memory implementation, based on a domain decomposition, requires 'decomposing' just the two arrays.
2. Memory for data structure arrays, solvers and graphics is allocated dynamically.
3. All logical operations have been separated into two stages: operations on nodes and operations on degrees of freedom for a node. In particular, elements contain all higher order nodes (mid-edge, mid-side and middle nodes), even for lower orders of approximation. Only the memory for the corresponding degrees of freedom is allocated dynamically as needed. This simplifies the logic and makes customization of the code for electromagnetics [6] easier.
4. The information about contraints is *reconstructed* locally, on element level, based on data structure arrays for the element and his father [1]. This represents a significant departure from the first two implementations:
   - reconstruction of constraints based on element neigbors [1],
   - explicit storing of the constraints in the data structure [3].

Compared with the 2D implementation [5], a number of additional changes have been introduced. We mention here the two most important ones:

- Procedures of breaking an element's interior, side and edge have been introduced and coded in separate routines. This has allowed, in particular, to reduce the breaking of an element into a sequence of successive breakings of the element interior, sides and edges. The same logic has been applied to coding p-refinements.

[1] Possibly grandfather or even greatgrandfather for double and triple constrained nodes

- The orientation of mid-edge, mid-base (valid for tetrahedra and prisms only), and mid-side nodes, has been identified as part of the connectivities information and coded as an extra attribute for an element.

The present, first version of the code, has only hexahedral elements in it. In its ultimate version, the code will include additionally prismatic and tetrahedral elements. Thus both the code and this manual will be subject of a continuous evolution that we hope to report in reasonable time intervals.

2 Model Problems

2.1 L^2-projection in 3D

Given a bounded domain Ω and an L^2-function $u(\boldsymbol{x})$ defined on Ω, we restrict ourselves to the simplest problem of projecting function u in the sense of the L^2-norm onto the finite element space. More precisely, if X_h denotes the *finite element space* consisting of functions v_h that are globally continuous and, over each element K, "live" in the corresponding *element space* $X(K)$, we want to solve the minimization problem:

$$\|u - u_h\| = \inf_{v_h \in X_h} \|u - v_h\| \tag{2.1}$$

Here $\| \ \|$ denotes the L^2-norm:

$$\|u\|^2 = \int_\Omega u(\boldsymbol{x})^2 \, d\boldsymbol{x} \ . \tag{2.2}$$

The problem admits a unique solution u_h that can be found by solving an equivalent variational equation:

$$\begin{cases} u_h \in X_h \\ \displaystyle\int_\Omega u_h v_h \, d\boldsymbol{x} = \int_\Omega u v_h \, d\boldsymbol{x} \quad \forall v_h \in X_h \end{cases} \tag{2.3}$$

In particular, if the projected function $u(\boldsymbol{x})$ is from space X_h itself, its projection u_h must coincide with the original function, and the corresponding L^2-error should be, up to machine accuracy, equal zero. This makes the L^2-projection problem a convenient tool for debugging the code, especially the constrained approximation package.

Setting up data for the problem

User has to provide *only* routine *l2proj/exact* evaluating, for a given point, value of the projected function.

2.2 Laplace (Poisson) equation in 3D

We are given a bounded domain Ω, with boundary Γ consisting of two disjoint parts, Γ_D (Dirichlet) and Γ_N (Neumann). We wish to solve the boundary-value problem:

$$\begin{cases} -\Delta u &= f & \text{in } \Omega \\ u &= u^D & \text{on } \Gamma_D \\ \dfrac{\partial u}{\partial n} &= g & \text{on } \Gamma_N \end{cases} \tag{2.4}$$

Here Δ denotes the Laplacian, $\partial u/\partial n$ is the normal derivative on the boundary, and f, u^D, g are functions defined on Ω, Γ_D, and Γ_N, respectively.

With appropriate regularity assumptions, the problem admits a unique solution u_h that can be found by solving an equivalent variational equation:

$$\begin{cases} u \in H^1(\Omega), u = u^D \text{ on } \Gamma_D, \\ \int_\Omega \boldsymbol{\nabla} u \boldsymbol{\nabla} v \, d\boldsymbol{x} = \int_\Omega f v \, d\boldsymbol{x} + \int_{\Gamma_N} g v \, dS \qquad \forall v \in H^1(\Omega), v = 0 \text{ on } \Gamma_D \end{cases} \tag{2.5}$$

The approximate solution is obtained then by solving an analogous problem in the FE spaces.

$$\begin{cases} u_h \in u_h^D + V_h \\ \int_\Omega \boldsymbol{\nabla} u_h \boldsymbol{\nabla} v_h \, d\boldsymbol{x} = \int_\Omega f v_h \, d\boldsymbol{x} + \int_{\Gamma_N} g v_h \, dS \quad \forall v_h \in V_h \end{cases} \tag{2.6}$$

Here u_h^D denotes a lift (extension) of a FE approximation (denoted with the same symbol) of Dirichlet data u^D, to the whole domain Ω [2], and V_h stands for the space of test functions satisfying the homogeneous Dirichlet boundary condition:

$$V_h = \{v_h \ : \ v_h = 0 \text{ on } \Gamma_D\} \tag{2.7}$$

Setting up data for the problem

There are two ways of inputing data.

- **Case 1: Known exact solution.** User has to provide *only* routine *laplace/exact* evaluating, for a given point, value of the solution, and its first and second order derivatives. The cooresponding data for the problem are then calculated in routine *laplace/coeff* by calling routine *exact*. The routine is also called in the graphics package for displaying the exact solution. That way we have minimized the number of changes in the code necessary to study different solutions.

- **Case 2: Exact solution unknown.** In this case, user has to provide routine *laplace/coeff* that returns functions f, g, u^D, himself. Routine *exact* must be faked then.

[2] In practice, finite element domain will differ from the original domain Ω.

3 The *hp* Finite Element Method on Regular Meshes

We recall [5] that the Finite Element Method is a special case of the Galerkin method and differs from other methods in the way the basis functions are constructed. Domain Ω is partitioned into disjoint subdomains called *finite elements*. In practice, the 3D elements have shapes of hexahedra, triangular prisms, or tetrahedra, possibly with curvilinear sides and edges. Next, for each element K, we introduce the corresponding *shape functions* ϕ_K which eventually are *glued* into the globally defined basis functions e_k in the Galerkin method. It is the construction of the basis functions that distinguishes the FEM from other Galerkin approximations. Referring to [5] for a discussion of 1D and 2D elements, we proceed directly with the definition of the 3D hexahedral master element of variable order.

3.1 Hexahedral master element

The element occupies the standard reference cube, $\hat{K} = [0,1]^3$. The element space of shape functions $X(\hat{K})$ is a subspace of $Q^{(p_x,p_y,p_z)}$, i.e. polynomials that are of order p_x, p_y, p_z in ξ_1, ξ_2, ξ_3, respectively. In order to be able to match in one mesh elements of different orders, we associate with each of the element sides a possibly different order of approximation $\boldsymbol{p}_i = (p_{ih}, p_{iv})$ where h and v refer to the *horizontal* and *vertical* orders of approximation for each of the six sides $i = 1, \ldots, 6$. The sides are denumerated in the order: bottom,top,front,right,rear,left, and the corresponding *local* horizontal and vertical directions for each side coincide with the relevant global directions ordered in the lexicographic order:

$$(1,2) \quad (1,2) \quad (1,3) \quad (2,3) \quad (1,3) \quad (2,3).$$

Similarly, with each of the twelve edges, we associate a possibly different order of approximation $p_j, j = 1, \ldots, 12$. We shall assume that the orders of approximation (p_h, p_v) for a side do not exceed the corresponding orders in the global directions, and similarly, the order of approximation for an edge, is less or equal than the corresponding orders for the two adjacent sides. The element space of shape functions is now identified as the subspace of $Q^{(p_x,p_y,p_z)}$, consisting of functions whose restrictions to sides $\hat{S}_i$ reduce to polynomials of (possibly smaller) order (p_{ih}, p_{iv}), and restrictions to edges $\hat{e}_j$ reduce to polynomials of (smaller) degree p_j,

$$X(\hat{K}) = \{\hat{u} \in Q^{(p_1,p_2,p_3)} \ : \ \hat{u}|_{\hat{S}_i} \in \mathcal{P}^{(p_{ih},p_{iv})}(\hat{S}_i), \quad \hat{u}|_{\hat{e}_j} \in \mathcal{P}^{p_j}(\hat{e}_j)\} \tag{3.1}$$

The element shape functions are constructed as *tensor products* of 1D shape functions defined in [5]. It is convenient to group them into subsets associated with the element vertices,

edges, sides and the element interior. Shape functions belonging to the same group have the same *connectivity*, i.e. the corresponding basis functions are built using the same logic. For instance, all basis functions associated with an interelement side "consist of" two shape functions corresponding to the two adjacent elements and the common side, whereas basis functions corresponding to an interior of an element "consist" of just one contributing shape function.

To facilitate the communication, we introduce the notion of abstract *nodes* that we associate with the element, see Fig. 1.

- eight vertex nodes: $\hat{\boldsymbol{a}}_j, j = 1, \ldots, 8$,
- twelve mid-edge nodes: $\hat{\boldsymbol{a}}_j, j = 9, \ldots, 20$,
- six mid-side nodes: $\hat{\boldsymbol{a}}_j, j = 21, \ldots, 26$,
- the middle node: $\hat{\boldsymbol{a}}_{27}$.

With each of the nodes, we associate the corresponding order of approximation, $p = 1$ for the vertex nodes, the edge order p_i for the i-th mid-edge node, the *anisotropic* order (p_{ih}, p_{iv}) for each mid-side node, and the *anisotropic* order of approximation (p_x, p_y, p_z) for the middle node. The corresponding shape functions are now defined as tensor products of the 1D shape functions $\hat{\chi}_i(\xi)$ in the following way.

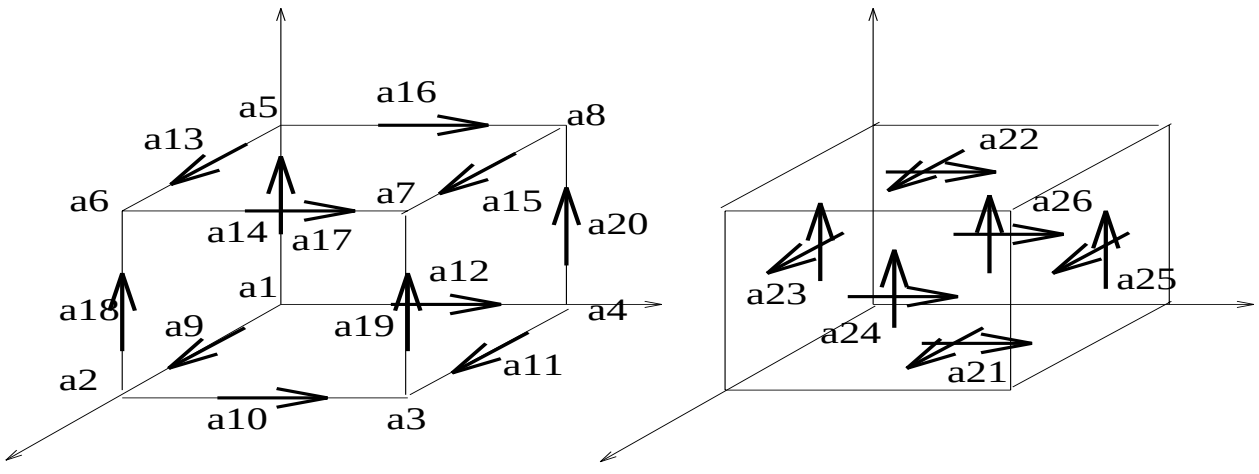

Figure 1: Hexahedral master element: (a) vertex and mid-edge nodes, (b) mid-side nodes

- one trilinear shape function for each of the vertex nodes,

$$
\begin{aligned}
\hat{\phi}_1(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3) \\
&= (1-\xi_1)(1-\xi_2)(1-\xi_3) \\
\hat{\phi}_2(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3) \\
&= \xi_1(1-\xi_2)(1-\xi_3) \\
\hat{\phi}_3(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3) \\
&= \xi_1\xi_2(1-\xi_3) \\
\hat{\phi}_4(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3) \\
&= (1-\xi_1)\xi_2(1-\xi_3) \\
\hat{\phi}_5(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3) \\
&= (1-\xi_1)(1-\xi_2)\xi_3 \\
\hat{\phi}_6(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3) \\
&= \xi_1(1-\xi_2)\xi_3 \\
\hat{\phi}_7(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3) \\
&= \xi_1\xi_2\xi_3 \\
\hat{\phi}_8(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3) \\
&= (1-\xi_1)\xi_2\xi_3 \ ,
\end{aligned}
\tag{3.2}
$$

- $p_i - 1$ shape functions for each of the mid-edge nodes,

$$
\begin{aligned}
\hat{\phi}_{9,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}(1-\xi_2)(1-\xi_3), \quad j=1,\ldots,p_1-1 \\
\hat{\phi}_{10,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_1(\xi_3) \\
&= \xi_1(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}(1-\xi_3), \quad j=1,\ldots,p_2-1 \\
\hat{\phi}_{11,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}\xi_2(1-\xi_3), \quad j=1,\ldots,p_3-1 \\
\hat{\phi}_{12,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_1(\xi_3) \\
&= (1-\xi_1)(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}(1-\xi_3), \quad j=1,\ldots,p_4-1 \\
\hat{\phi}_{13,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}(1-\xi_2)\xi_3, \quad j=1,\ldots,p_5-1 \\
\hat{\phi}_{14,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_2(\xi_3) \\
&= \xi_1(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}\xi_3, \quad j=1,\ldots,p_6-1 \\
\hat{\phi}_{15,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}\xi_2\xi_3, \quad j=1,\ldots,p_7-1 \\
\hat{\phi}_{16,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_2(\xi_3) \\
&= (1-\xi_1)(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}\xi_3, \quad j=1,\ldots,p_8-1 \\
\hat{\phi}_{17,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_{2+j}(\xi_3) \\
&= (1-\xi_1)(1-\xi_2)(1-\xi_3)\xi_3(2\xi_3-1)^{j-1}, \quad j=1,\ldots,p_9-1 \\
\hat{\phi}_{18,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_{2+j}(\xi_3) \\
&= \xi_1(1-\xi_2)(1-\xi_3)\xi_3(2\xi_3-1)^{j-1}, \quad j=1,\ldots,p_{10}-1 \\
\hat{\phi}_{19,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_{2+j}(\xi_3) \\
&= \xi_1\xi_2(1-\xi_3)\xi_3(2\xi_3-1)^{j-1}, \quad j=1,\ldots,p_{11}-1 \\
\hat{\phi}_{20,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_{2+j}(\xi_3) \\
&= (1-\xi_1)\xi_2(1-\xi_3)\xi_3(2\xi_3-1)^{j-1}, \quad j=1,\ldots,p_{12}-1\,,
\end{aligned}
\tag{3.3}
$$

- $(p_{ih}-1)(p_{iv}-1)$ *side bubble shape functions* for each of the mid-side nodes,

$$
\begin{aligned}
\hat{\phi}_{21,jk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_{2+k}(\xi_2)\hat{\chi}_1(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}(1-\xi_2)\xi_2(2\xi_2-1)^{k-1}(1-\xi_3) \\
&\quad j=1,\ldots,p_{13h}-1,\ k=1,\ldots,p_{13v}-1 \\
\hat{\phi}_{22,jk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_{2+k}(\xi_2)\hat{\chi}_2(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}(1-\xi_2)\xi_2(2\xi_2-1)^{k-1}\xi 3 \\
&\quad j=1,\ldots,p_{14h}-1,\ k=1,\ldots,p_{14v}-1 \\
\hat{\phi}_{23,jk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_{2+k}(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}(1-\xi 2)(1-\xi_3)\xi_3(2\xi_3-1)^{k-1} \\
&\quad j=1,\ldots,p_{15h}-1,\ k=1,\ldots,p_{15v}-1 \\
\hat{\phi}_{24,jk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_{2+k}(\xi_3) \\
&= \xi_1(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}(1-\xi_3)\xi_3(2\xi_3-1)^{k-1} \\
&\quad j=1,\ldots,p_{16h}-1,\ k=1,\ldots,p_{16v}-1 \\
\hat{\phi}_{25,jk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_{2+k}(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}\xi 2(1-\xi_3)\xi_3(2\xi_3-1)^{k-1} \\
&\quad j=1,\ldots,p_{17h}-1,\ k=1,\ldots,p_{17v}-1 \\
\hat{\phi}_{26,jk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_{2+k}(\xi_3) \\
&= (1-\xi_1)(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}(1-\xi_3)\xi_3(2\xi_3-1)^{k-1} \\
&\quad j=1,\ldots,p_{18h}-1,\ k=1,\ldots,p_{18v}-1
\end{aligned}
\tag{3.4}
$$

- $(p_x-1)(p_y-1)(p_z-1)$ *bubble shape functions* for the middle node,

$$
\begin{aligned}
\hat{\phi}_{27,ijk}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+i}(\xi_1)\hat{\chi}_{2+j}(\xi_2)\hat{\chi}_{2+k}(\xi_3) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{i-1}(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}(1-\xi_3)\xi_3(2\xi_3-1)^{k-1} \\
&\quad i=1,\ldots,p_x-1,\ j=1,\ldots,p_y-1,\ k=1,\ldots,p_z-1\,.
\end{aligned}
\tag{3.5}
$$

REMARK 1 Please note that, contrary to the 2D implementation, we do not differentiate between the first and third, the second and fourth, etc. edges. When restricted to an element edge, the 3D shape functions reduce to 1D shape functions, with the local coordinate ξ oriented consistently with the parallel ξ_i axis. Similarly, when restricted to an element side, the 3D shape functions reduce to 2D bubble shape functions, with the local coordinates ξ_h, ξ_v oriented consistently with the parallel ξ_i axes. ∎

We emphasize the abstract character of the "nodes" introduced above. They are merely an abstraction for the element vertices, edges, sides, and its interior, and should not be confused with the classical notion of the Lagrange or Hermite nodes.

Finally, we introduce the notion of the *hp-interpolation procedure*. The idea is a direct generalization of the 1D, and 2D *hp*-interpolations defined in [5]. Given a continuous function $\hat{u}(\xi_1, \xi_2, \xi_3)$ defined over the master element, we define its *hp*-interpolant as the sum of four contributions:

$$\hat{u}_{hp} = \hat{u}^1_{hp} + \hat{u}^2_{hp} + \hat{u}^3_{hp} + \hat{u}^4_{hp} \tag{3.6}$$

where

- $\hat{u}^1_{hp}$ is the standard trilinear interpolant corresponding to the vertex nodes:

$$\hat{u}^1_{hp}(\boldsymbol{\xi}) = \sum_{i=1}^{8} \hat{u}(\hat{\boldsymbol{a}}_i)\hat{\phi}_i(\boldsymbol{\xi}) \,, \tag{3.7}$$

- $\hat{u}^2_{hp}$ is obtained by projecting the difference of the original function and its trilinear interpolant onto the span of shape functions associated with the mid-edge nodes. More precisely,

$$\hat{u}^2_{hp} = \sum_{i=1}^{12} \hat{u}^{2i}_{hp} \tag{3.8}$$

  where

$$\hat{u}^{2i}_{hp}(\boldsymbol{\xi}) = \sum_{j=1}^{p_i - 1} u_{ij}\hat{\phi}_{8+i,j}(\boldsymbol{\xi}) \tag{3.9}$$

  and the coefficients u_{ij} are determined solving the system of equations:

$$\sum_{j=1}^{p_i-1} u_{ij} \int_{\hat{e}_i} \frac{d\hat{\phi}_{8+i,j}}{d\xi}\frac{d\hat{\phi}_{8+i,k}}{d\xi}\, d\xi = \int_{\hat{e}_i} (\frac{d\hat{u}}{d\xi} - \frac{d\hat{u}^1_{hp}}{d\xi})\frac{d\hat{\phi}_{8+i,k}}{d\xi}\, d\xi \,, \tag{3.10}$$

- $\hat{u}^3_{hp}$ is obtained by projecting the difference $\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp}$ onto the span of mid-side bubble shape functions associated with the mid-side nodes, More precisely,

$$\hat{u}^3_{hp} = \sum_{i=1}^{6} \hat{u}^{3i}_{hp} \tag{3.11}$$

where

$$\hat{u}^{3i}_{hp} = \sum_{j=1}^{p_{ih}-1} \sum_{k=1}^{p_{iv}-1} u_{jk} \hat{\phi}_{20+i,jk} \tag{3.12}$$

and the coefficients u_{jk} are determined by solving the system of equations:

$$\begin{aligned} &\sum_{j=1}^{p_{ih}-1} \sum_{k=1}^{p_{iv}-1} u_{jk} \int_{\hat{S}_i} \boldsymbol{\nabla} \hat{\phi}_{20+i,jk} \boldsymbol{\nabla} \hat{\phi}_{20+i,lm} \, d\boldsymbol{\xi} = \int_{\hat{S}_i} \boldsymbol{\nabla} (\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp}) \boldsymbol{\nabla} \hat{\phi}_{20+i,lm} \, d\boldsymbol{\xi} \\ &\quad l = 1, \ldots, p_{ih} - 1, m = 1, \ldots, p_{iv} - 1 \, . \end{aligned} \tag{3.13}$$

where $\boldsymbol{\nabla}$ denotes the gradient corresponding to the *local* side coordinates (ξ_h, ξ_v) and $d\boldsymbol{\xi} = d\xi_h d\xi_v$;

- and, finally, $\hat{u}^4_{hp}$ is obtained by projecting the difference $\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp} - \hat{u}^3_{hp}$ onto the span of bubble shape functions associated with the middle node,

$$\left\{ \begin{aligned} &\hat{u}^4_{hp} = \sum_{i=1}^{p_x-1} \sum_{j=1}^{p_y-1} \sum_{k=1}^{p_z-1} u_{ijk} \hat{\phi}_{27,ijk} \\ &\sum_{i=1}^{p_x-1} \sum_{j=1}^{p_y-1} \sum_{k=1}^{p_z-1} u_{ijk} \int_{\hat{K}} \boldsymbol{\nabla} \hat{\phi}_{27,ijk} \boldsymbol{\nabla} \hat{\phi}_{27,lmn} \, d\boldsymbol{\xi} = \int_{\hat{K}} \boldsymbol{\nabla} (\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp} - \hat{u}^3_{hp}) \boldsymbol{\nabla} \hat{\phi}_{27,lmn} \, d\boldsymbol{\xi} \\ &\quad l = 1, \ldots, p_x - 1, m = 1, \ldots, p_y - 1, n = 1, \ldots, p_z - 1 \, . \end{aligned} \right. \tag{3.14}$$

The *hp*-interpolant can be viewed as a Galerkin solution of the Poisson equation with the Dirichlet boundary data set to the two-dimensional *hp*-interpolant of the original function.

3.2 Parametric element

We use the standard procedure to define parametric (deformed) hexahedra. Given a bijective map

$$\boldsymbol{x}_K : \hat{K} \to K \tag{3.15}$$

from master hexahedron onto an element K, we define the element space of shape functions as a collection of compositions of inverse $\boldsymbol{x}_K^{-1}$ and the master element shape functions,

$$X(K) = \{u = \hat{u} \circ \boldsymbol{x}_K^{-1} \ : \ \hat{u} \in X(\hat{K})\}. \tag{3.16}$$

Accordingly, the element shape functions are defined as:

$$\phi_i(\boldsymbol{x}) = \hat{\phi}_i(\boldsymbol{\xi}), \text{ where } \boldsymbol{x}_K(\boldsymbol{\xi}) = \boldsymbol{x}. \tag{3.17}$$

For each element K, we shall speak about its vertex, mid-edge, mid-side, and middle nodes, understood again simply as an abstraction for the element vertices, edges, sides and interior.

Finally, similarly as in 1D and 2D, we define the *hp*-interpolation procedure for curved elements via the master element. Given a continuous function $u(\boldsymbol{x})$ defined over an element K, we consider its counterpart $\hat{u}(\boldsymbol{\xi})$ defined over the corresponding master element,

$$\hat{u}(\boldsymbol{\xi}) = u(\boldsymbol{x}_K(\boldsymbol{\xi}))\,, \tag{3.18}$$

we interpolate next $\hat{u}$ on the master element, and come back to the actual element K,

$$u_{hp}(\boldsymbol{x}) = \hat{u}_{hp}(\boldsymbol{\xi}), \quad \boldsymbol{x}_K(\boldsymbol{\xi}) = \boldsymbol{x} \tag{3.19}$$

where $\hat{u}_{hp}$ denotes the *hp*-interpolant of $\hat{u}$ over the master element. Once again, the important point is that the calculation of the interpolant takes place on the master element.

We restrict ourselves to the *isoparametric* deformations only, i.e. we assume that the map $\boldsymbol{x}_K$ lives in the corresponding space of shape functions for the master element. More precisely,

$$\boldsymbol{x}_K(\boldsymbol{\xi}) = \sum_j \boldsymbol{x}_{Kj} \hat{\phi}_j(\boldsymbol{\xi}) \tag{3.20}$$

Here j is the index denumerating shape functions of the master element, and $\boldsymbol{x}_{Kj}$ denotes the corresponding *vector-valued* geometry degrees-of-freedom. Note that only the degrees-of-freedom corresponding to the vertex nodes have the interpretation of the vertex nodes coordinates. If the map $\boldsymbol{x}_K$ is affine (in particular, all geometry d.o.f. except the vertex nodes coordinates are zero), the parametric element is called *affine element*, and the corresponding shape functions are polynomials.

3.3 Finite element space. Construction of basis functions

Let domain Ω be partitioned into hexahedral elements. We shall assume that no approximation of the boundary is necessary, i.e. that the boundary is at most polynomial and it

can be represented exactly with isoparametric elements of sufficiently high order. We shall also assume that the mesh is *regular* in the sense that the intersection of any two elements is either empty, or it consists of a single vertex node, a (whole) common edge, or a (whole) common side. Otherwise the mesh will be called *irregular*. Irregular meshes appear naturally as a result of h refinements discussed in the next section. We now define the finite element space X_h as the collection of continuous functions whose restrictions to elements K live in the element spaces of shape functions,

$$X_h = \{u \ : \ u \text{ is continuous and } u|_K \in X(K), \text{ for every element } K \text{ in the mesh}\} \quad (3.21)$$

The global basis functions are obtained by "gluing together" element shape functions. The actual procedure varies for vertex, mid-edge, mid-side, and middle nodes. For vertex nodes, the procedure is standard. For a given vertex node $\boldsymbol{a}_i$, the restriction of the corresponding basis function e_i to a contributing (adjacent) element K coincides with one of the element vertex shape functions:

$$e_i|_K = \phi_k \quad (3.22)$$

Connectivity $i = i(k, K)$ equals the number of the global vertex node corresponding to element K and its vertex shape function k.

For mid-edge nodes, the procedure is the same as in 2D case [5]. For a given mid-edge node $\boldsymbol{a}_i$ and corresponding j-th basis function $e_{i,j}$, the restriction of the basis function to a contributing (adjacent) element K reduces to one of the element edge shape functions $\phi_{8+k,j,K}$, premultiplied possibly with a *sign factor* $c_{k,j,K}$,

$$e_{i,j}|_K(\boldsymbol{x}) = c_{k,j,K}\phi_{8+k,j,K}(\boldsymbol{x}). \quad (3.23)$$

We view the *connectivity* at the nodal level; $i = i(k, K)$ equals the number of the global (mid-edge) node corresponding to element K and its k-th node. Note that indices j, indicating the number of the nodal shape function are the same, i.e. the j-th shape function of the k-th element local node corresponds to the j-th shape function associated with the global i-th node.

The necessity of introducing the sign factor for mid-edge shape functions results from the fact that the local edge parametrizations for two adjacent elements may be opposite to each other. Consequently, the edge shape functions of odd degree corresponding to the elements *do not match each other*, and one of them has to be premultiplied by a -1 factor.

More precisely, each of the edges is assumed to have a *global orientation*. If the local orientation of an element edge is now consistent with the global one, the corresponding sign factor will be 1, otherwise it will be -1.[3] For the initial mesh, we use the standard rule

[3]For odd degree shape functions only. For even degree shape functions the sign factor is always equal 1.

and assume that the global orientation of edges follows from the numbering of its vertex endpoints nodes - the edge will always be oriented from the vertex node with a smaller number to the vertex node with a bigger number. The corresponding sign factors relating local and global orientations for an element are stored then as one of the attributes of the element to be discussed in the next section. During mesh refinements, edges are broken, and the global orientation for the new mid-edge nodes is assumed to coincide with the orientation of the *parent mid-edge node*. The formal definition of the sign factor is now as follows:

$$c_{k,j,K} = \begin{cases} -1 & \text{if the local orientation of the } k\text{-th edge of element } K \text{ is opposite} \\ & \text{to the global one } \textit{and } j \text{ is even} \\ 1 & \text{otherwise} \end{cases} \tag{3.24}$$

Constructing global basis functions for a mid-side node $\boldsymbol{a}_i$ is more complicated. First, for each side we introduce a *global* side system of coordinates consisting of *horizontal* and *vertical* axes. Given an element adjacent to the side, and the corresponding *local* system of coordinates for the side, we can identify eight possible different orientations of the local side axes with respect to the global ones. The information about the orientation of the local mid-side node coordinates wrt the global one is again stored in a compact form in the data structure arrays and can be retrieved by calling routine *datstrb/nodorieb* which, for each node, returns a flag and two sign factors. The flag indicates whether the global and local directions for the side have been switched or not. If the directions of global and local axes are the same (up to the orientations of the axes !), the restriction of jk-th side basis function to the element equals the corresponding element shape function $\phi_{20+l,jk,K}$ premultiplied by two sign factors $c^1_{l,m,K}, c^2_{l,n,K}$, returned by routine *datstrb/nodorieb* and indicating whether the sense of direction for the local and global axes are the same or opposite to each other:

$$e_{i,jk}|_K = c^1_{l,m,K} c^2_{l,n,K} \phi_{20+l,mn,K}. \tag{3.25}$$

Here l denotes the local number of the side withing the considered element. In the case when the global and local axes have been switched, the corresponding shape functions indices get switched as well:

$$e_{i,jk}|_K = c^1_{l,m,K} c^2_{l,n,K} \phi_{20+l,nm,K}. \tag{3.26}$$

The global mid-side axes are defined first for the initial mesh, using the side vertex nodes numbers and the following rules:

- the origin is located at the vertex with the smallest node number;
- the horizontal axis connects the origin with the one from neigboring vertex nodes that has a smaller node number;

- the second axis points to the remaining neigboring vertex node.

During refinements, when a side is "broken", it gives rise to new mid-side nodes and new mid-edge node(s). The orientation of the new mid-side nodes is identical with the orientation of the parent mid-side node; the orientation of the new mid-edge nodes coincides with the orientation of the corresponding parallel axis of the parent mid-side node.

Finally, the global basis functions corresponding to a middle node coincide with the corresponding bubble element shape function and no assembling is necessary.

The calculation of the global stiffness matrix and load vector is the same as in 2D, see [5] for a detailed description.

3.4 Data structure in FORTRAN 90

We introduce two *user-defined structures* (see *module/data_structure*):

- type *node*,
- type *element*.

The attributes of a node include: node type (a character indicating whether the node is a vertex, mid-edge, mid-side or a middle node), integer order of approximation, integer boundary condition flag, an integer geometric modeling interface flag for future use, and two real arrays, *coord*, containing geometrical degrees-of-freedom, and $dofs$, containg the "actual" degrees-of-freedom. Both the geometry and actual d.o.f. are allocated dynamically, dependently upon the order of approximation for the node. Note the following details:

- the integer specifying order for a mid-side node is actually a nickname defined as

$$order = p_h * 10 + p_v \tag{3.27}$$

  where p_h, p_v are the *horizontal* and the *vertical* orders of approximation for the node[4];

- similarly, the order of approximation for a middle node is a nickname defined as

$$order = p_x * 100 + p_y * 10 + p_z \ ; \tag{3.28}$$

[4] We refer here to the *global* side system of coordinates.

- both *coord* and $dofs$ are defined as arrays with two indices. The first index of *coord* corresponds to the number of coordinates and varies between 1 and 3. The first index of array $dofs$ indicates a component of the solution, and for a single equation problem like the L^2projection or the Laplace equation, is always equal one.

Out of the list of the attributes of an element, listed in module *moduli/data_structure*, we discuss for a moment only four:

- integer array *nodes* contains the twenty seven node numbers of the element (the connectivity info);
- integer array orient stores orientations for the element mid-edge (the first entry) and mid-side nodes (the third entry), the second entry is reserved for a future use for triangular facets (mid-base nodes);
- integer array *neig* contains numbers of six neigbors of the element (across its faces), a zero entry indicates that the side is adjacent to a boundary of the domain;
- integer *bcond* is a nickname encoding boundary conditions flags for the element sides:

$$bcond = bc6 * 10000 + bc5 * 10000 + bc4 * 1000 + bc3 * 100 + bc2 * 10 + bc1 \quad (3.29)$$

  where $bc1, \ldots, bc6$ are one-digit flags for the element sides.

The remaining attributes are related to h-refinements, which are discussed in the next section.

The entire information about a mesh is now stored in two allocatable arrays, ELEMB and NODEB, as declared in the data structure module. The module also includes a declaration for a number of integer attributes of the mesh. Two of them are relevant at the moment: $NRELEB$ - total number of *active* elements in the mesh, and $NRNODB$ - total number of nodes in the mesh.

3.5 The element routine

The procedure for evaluating the element stiffness matrix and load vector is the same as in the 2D implementation [5].

3.6 Modified element

As discussed earlier, assembling of element matrices into the global matrices is accompanied for some shape functions by a change of sign and possible renumeration (mid-side nodes). It

is convenient to separate these two operations, the sign change and the renumeration, from the assembling procedure, by introducing the notion of a *modified element*.[5]

The information about the modified element includes:

- a list of element vertex nodes: $Nod1(i), i = 1, \ldots, Nrvert$,
- a list of element higher order nodes: $Nod2(i), i = 1, \ldots, Nrnod$,
- the corresponding number of shape functions (d.o.f) per node: $Ndof2(i), i = 1, \ldots, Nrnod$,
- the modified element load vector: $Bload(k), k = 1, \ldots, Nrdof$,
- the modified element stiffnes matrix: $Astiff(k, l),\ k, l = 1, \ldots, Nrdof$.

The shape functions are ordered following the order of nodes and the corresponding order of shape functions for each of the nodes corresponding to the *global* orientation. The total number of shape functions (degrees-of-freedom) for the element, $Nrdof$, is obtained by summing up the numbers of shape functions for each node:

$$Ndof = Nrvert + \sum_{i=1}^{Nrnod} Ndof2(i) \tag{3.30}$$

For regular meshes discussed in this section, the lists of vertex and higher order nodes are organized in the same order as they are listed in the data structure array $ELEMB(nel)\%nodes$. If for a node, the corresponding number of shape functions is zero, the node is skipped form the list. The corresponding matrices are obtained by multiplying the local matrices by the sign factors discussed earlier. More precisely, for each *local* d.o.f. k, the following values are returned from routine *constrb/logunbr*:

- number of modified element d.o.f. connected to the k-th local d.o.f. (just one in this case, more relevant for a future use):

$$Nrcon(k) = 1$$

- numbers (just one) of the connected modified d.o.f.:

$$nac(j, k), k = 1, Nrcon(k)$$

- the corresponding multiplicative factors related to the sign factors discussed earlier:

$$constr(j, k), j = 1, \ldots, Nrcon(k)$$

The calculation of the modified element matrices is done in routine *constrb/celem* using the standard algorithm discussed in [5].

[5] The need for this separation will be more clear for h-adapted meshes discussed in the next section.

Assembling of global matrices is standard and identical with that in the 2D code.

Interface with a frontal solver is done in the same way as in the 2D code.

3.7 Interfacing with the Geometric Modeling Package (GMP)

Initial mesh generation

See again [5] for the discussion of a simple multiblock initial mesh generator based on the *hp*-interpolation procedure and the GMP package [2]. Sample input files, specifying input for GMP and the mesh generator, can be found in *files/inputs*.

An interface with *GMP*, defined in module *module_GMP*, is initiated during the mesh generation. It allows to find for each element the corresponding geometrical block, and location of the element within the block, specified in terms of integer coordinates. This information is crucial when generating new nodes during mesh refinements.

p-adaptivity

Finally, we discuss the possibility of *p*-refinements (unrefinements). Since the modifications do not introduce new or delete existing nodes, the corresponding changes in the data structure arrays are minimal. Order of approximation for an element *nel* can be modified (increased or decreased) by invoking routine *meshmods/enrichb*. The routine enforces the *minimum rule*, i.e. the order for an edge or side common for a number of elements is fixed to the minimum of the orders for the neighboring elements. The actual changes in data structure arrays are executed in routine *meshmodb/nodmodb*. Please try to modify the order for an element using the interactive routine for mesh modifications *graph_body/meshb_new* in order to verify the discussed rules.

When the order of approximation of a node is increased, the corresponding new geometry d.o.f. are initiated with zeros. Similarly, if the new order of approximation is smaller than the old one, the redundant d.o.f. are simply deleted and the old ones are left without any change. In order to update the d.o.f. using the interface with *GMP*, one has to call explicitly routine *hp_interp/update_gdof*. The routine loops through a list of elements to be updated and regenerates all geometry d.o.f. using the same *hp*-interpolation procedure as during the initial mesh generation.

4 The hp Finite Element Method on h-Refined Meshes

4.1 Introduction. The h-refinements

The present anisotropic h-refinment strategy is an extension of the anisotropic h-refinements strategy for quads presented in [5]. A father element is always divided into two element-sons by slashing the father with a plane *across* the first, second or third axis. Accordingly, we shall speak about the *refirement kind* $nref$ being equal 1,2 or 3, respectively. As a result of h-refinements, constrained nodes appear in the mesh. An element with no constrained nodes is said to be *unconstrained*. In particular, all elements in the initial mesh are unconstrained. The essence of the mesh refinement strategy discussed next is that an unconstrained element can be refined first across, say, the first axis, next its sons can be refined, say, across the second axis, and finally, the resulted *grandsons* can be refined across the remaining third axis - all *without* any refinements of neighbors of the original element. This means that through the sequence of consequtive $h2$-refinements, an element can be divided into four grandsons (the $h4$-refinement) or eight greatgrandsons (the $h8$-refinement). The possibility of performing the consequtive $h2$-refinements without refining neighbors forces, similarly to the 2D case, the appearance of *double* and *triple constrained* nodes.

Breaking of an element consists of three distinctively different operations:

- breaking of the element interior,
- breaking of the element sides or, more precisely, the interiors of the sides,
- breaking of the element edges or, more precisely, the interiors of the edges.

Alternatively, we could think of breaking the element middle, mid-side, and mid-edge nodes. Let us discuss these operations in more detail.

Breaking the interior of an element. Geometrically, the element is sectioned into two sons with a plane parallel to one of the axes ($ref\_kind = 1, 2, 3$). In terms of the finite element discretization, the original element finite element space defined in the previous section is replaced with the space of *continuous* functions that are unions of polynomials defined over the element-sons, of the same order as for the father-element. However, over the boundary of the father-element, these piecewise polynomials reduce to the polynomials of the original element, i.e. the whole boundary of the father-element *remains unbroken*. Such an approximation is possible because of the constrained approximation technique to be discussed later in this section. Finally, from the data structure point of view, breaking

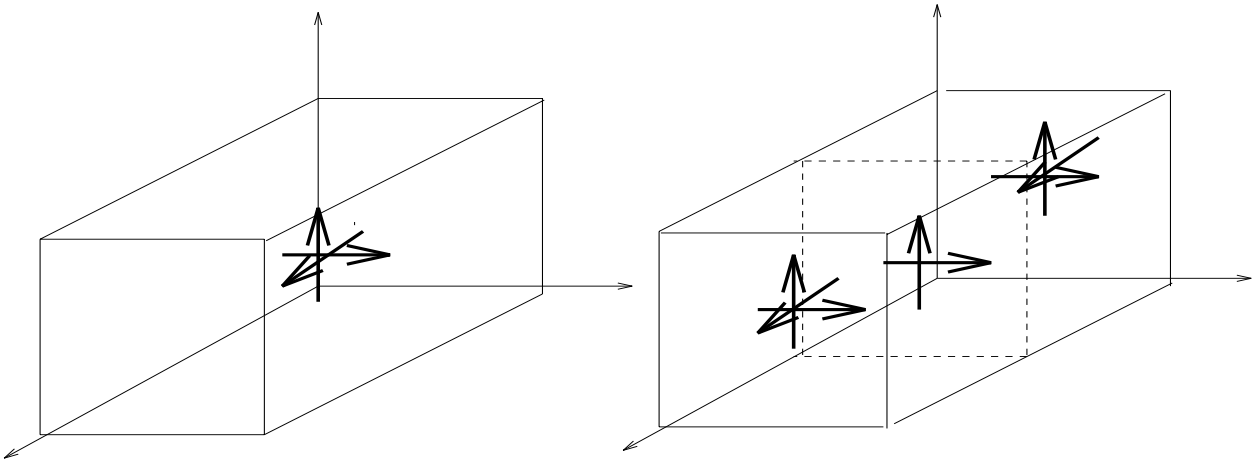

Figure 2: Breaking of an element interior. Two new middle nodes and one mid-side node are generated

of the element interior requires the generation of data for the two element-sons, generation of two new middle nodes and one mid-side node shared by the two sons. The operation of breaking of an element interior is illustrated in Fig 2.

Breaking a side. Once the interiors of two elements adjacent to a side have been broken, we proceed by breaking (the interior of) the side. A side can be broken into two smaller sides, either *horizontally* or *vertically* [6], or it can be broken *directly* into four smaller sides. The necessity of introducing the possibility of breaking the side directly into four sub-sides is related to the particular implementation of the constrained approximation and it is not related to the geometrical issues. When a side is to be broken, two situations may occur:

- refinement histories of the neighboring elements are compatible with each other, comp. Fig. 3,
- refinement histories of the neighboring elements are *incompatible* with each other.

The incompatibility occurs naturally when the interiors of two neighboring elements sharing a side are broken in two different directions, comp. Fig. 4. At this point, the side cannot be broken and both element-sons are *constrained* to the *big side* shared by their fathers. Eventually, the interiors of the sons on one side can be broken in the other direction, see Fig. 4 and, from the geometrical point of view, it should be possible to break the side. Due

[6] Recall that the horizontal and vertical directions refer here to the global side system of coordinates and have nothing to do with the actual geometry

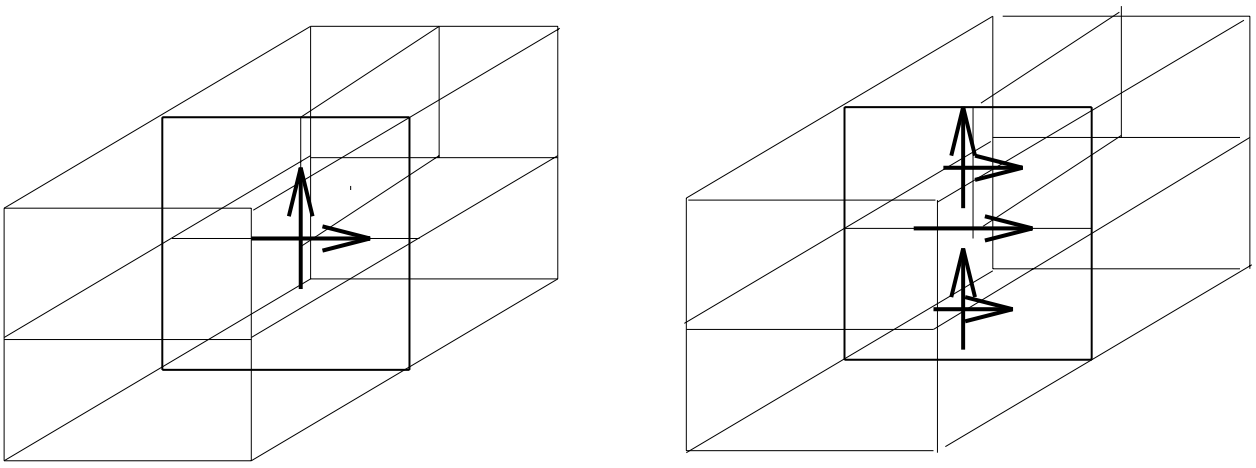

Figure 3: Breaking of an element side. Compatible refinements. Two new mid-side nodes and one mid-edge node are generated

to the fact, however, that in our implementation of constrained approximation, an element is always *constrained to its father*, even at this moment *the side cannot be broken*. Only eventually, when the all element-sons adjacent to the side are broken, the side is broken directly into *four smaller sides*.

Again, from the discretization point of view, breaking the side means replacing the original polynomial function with a new piecewice polynomial representation. Along the side edges, however, the approximation still coincides with the original polynomial representation, i.e. the edges *remain unbroken*. Finally, in terms of data structure changes, breaking a side means introducing two (or four) new mid-side nodes and one (four) new mid-edge nodes.

Breaking an edge. An edge can be broken, if all sides adjacent to the edge have been broken. We proceed then by switching from polynomial to piecewise polynomial discretization along the edge, i.e. introduce a new vertex node and two new mid-edge nodes, comp. Fig. 5.

Identifying the logical operations of breaking the interior of an element, a side, or an edge, has been very essential from the coding point of view and the overall logical structure of the code. We emphasize that these operations are *not* merely stages of the routine breaking an element but they represent coherent modifications of the mesh and the corresponding discretization. In particular, after any of the changes, we can compute with the resulted mesh.

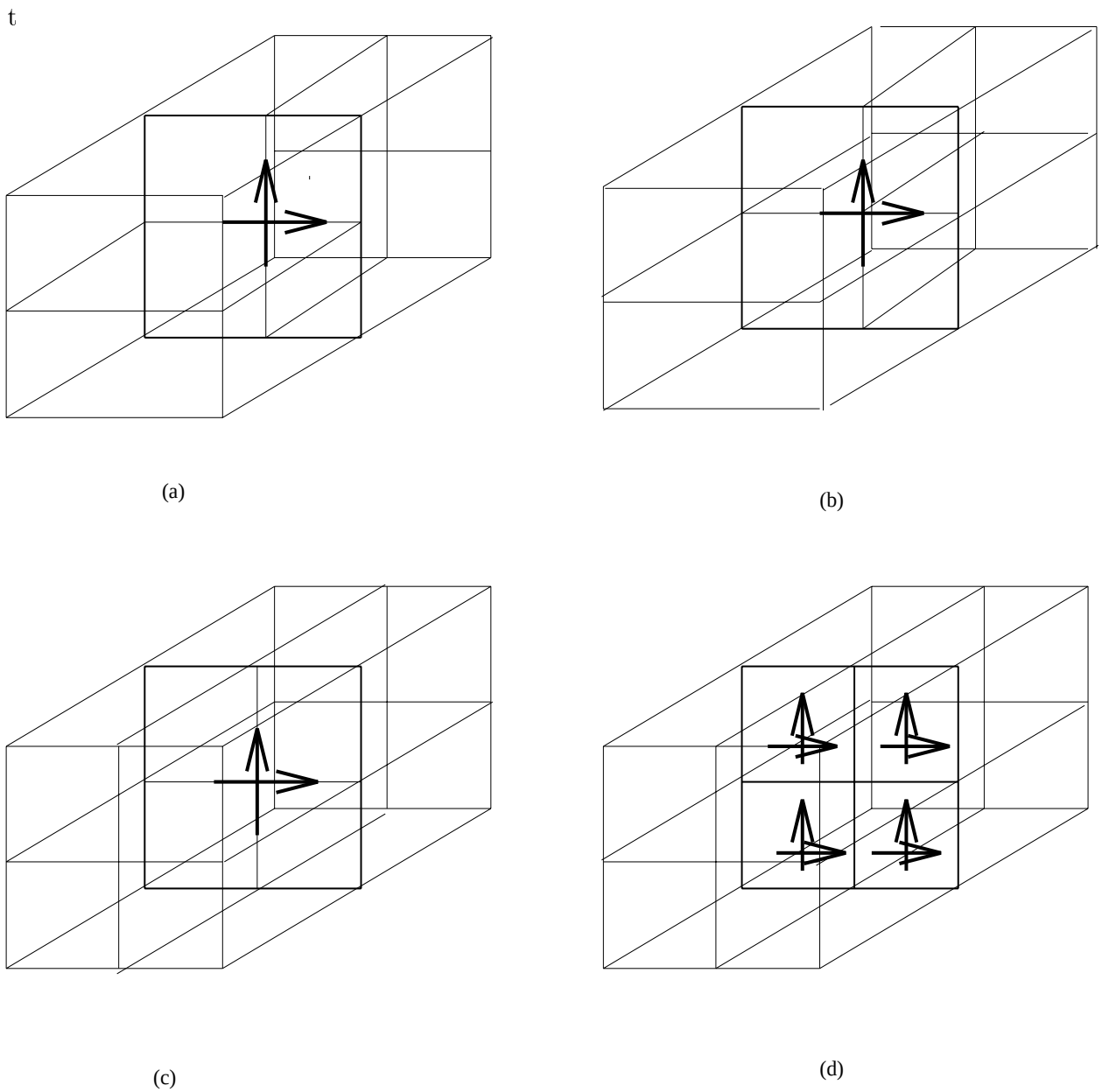

Figure 4: Breaking of an element side. Incompatible refinements. (a) Two neighboring elements have been broken in opposite directions. The side cannot be broken. (b) Sons of the rear element have been broken. The refinements of elements adjacent to the side are *geometrically compatible* but the histories of refinement are incompatible. The side remains unbroken. (c) Sons of the front element have been broken as well. The side can be broken now. (d) The side is broken into four smaller sides. Four new mid-side nodes and mid-edge nodes are generated

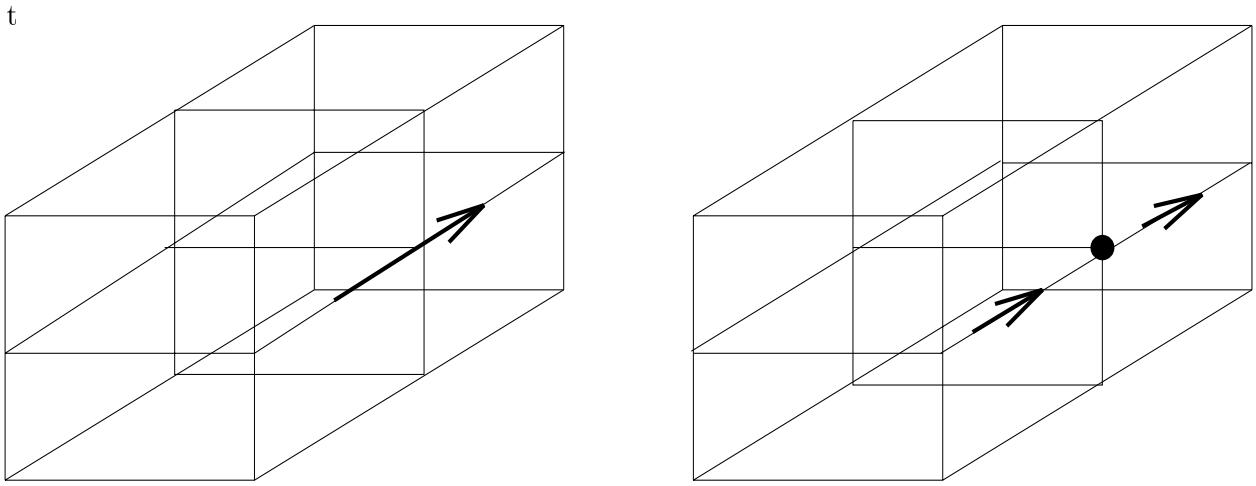

Figure 5: Breaking of an element edge. Two new mid-edge nodes and one vertex node are generated

4.2 The Anisotropic Mesh Refinements Algorithm

In effort to avoid the multiple constrained nodes, a generalization of the 2D strategy for quads [5] has been worked out.

The main idea of the algorithm is very simple. Before we attempt to refine an element, we identify its youngest ancestor that has resulted from the same refinement kind as the requested one. That ancestor has to be unconstrained. In practice that means that we have to check the element's refinement history. For instance, if we want to refine an element across the first axis ($ref\_kind = 1$ and its refinement history starts also from one, then the element must be unconstrained, before we proceed with the refinement. If the history is (2,1) (i.e. the element's father was slashed across the second axis but the grandfather across the first axis) or (3,1), then the father of the element must be unconstrained. Finally, if the history is (2,3,1) or (3,2,1) then the grandfather of the element must be unconstrained.

Once the ancestor that must be unconstrained has been identified, we eliminate its constrained nodes by refining its 'big' neighbors. The routine proceeds in two steps.

- We first localize 'big sides' constraining the element and attempt to break them, consistently with the procedure of breaking a side discribed above. This forces refinements of 'big elements' adjacent to the 'big sides'.
- Once all the 'big sides' have been eliminated, we identify all 'big edges' adjacent to the element, and proceed by breaking 'big elements' adjacent to the 'big edges'.

Finally, after the appropriate element ancestor's constrained nodes have been eliminated, we break the required element. The formal algorithm looks as follows.

Arguments in:

$Nel0$ element number

$Nref0$ refinement kind

= 1 for a refinement across the first axis[7]

= 2 for a refinement across the second axis

= 3 for a refinement across the third axis

Local variables:

$nela(*)$ a waiting list of elements to be refined

$nrefa(*)$ the corresponding refinement kinds

set $nel = Nel0, nref = Nref0$

10 continue

if nel is an unconstrained element go to 20

identify the element ancestor $nel1$ that must be unconstrained

for each 'big side' of element $nel1$

identify a neighbor 'neln' of the side that has to be broken first, and the corresponding refinement kind $nrefn$ to break the side;

store the current element and its refinement kind on the waiting list;

set $nel = neln, nref = nrefn$ and go to 10;

endfor

for each 'big edge' of element $nel1$

identify a neighbor 'neln' of the edge that has to be broken first, and the corresponding refinement kind $nrefn$ to break the edge;

store the current element and its refinement kind on the waiting list;

set $nel = neln, nref = nrefn$ and go to 10;

endfor

20 refine the element nel performing the $nref$ refinement

if the waiting list is empty then

stop

else

pick the next $nel, nref$ from the list and go to 10
endif

4.3 The Modified Anisotropic Mesh Refinements Algorithm

A disadvantage of the algorithm described above is the presence of double and triple constrained nodes in the mesh that may slow down the evaluation of the modified element matrices and complicates the overall structure of the discretization. Bearing in mind that the anisotropic refinements are needed mostly in regions where the solution is mostly one-dimensional (boundary layers), we propose a simplified version of the algorithm that avoids using multiple constrained nodes. The modified algorithm is identical with the previous one, except for the refinement part (after label 20). Three situations may occur there.

- Element $nel1 = nel$. Element nel is refined into two sons as in the original algorithm.
- Element $nel1$ is the father of element nel. *Both* sons (one of them is element nel) of element $nel1$ are refined in the requested (common) direction of refinement.
- Element $nel1$ is the grandfather of element nel. *All four* grandsons (one of them is element nel) of element $nel1$ are refined in the requested (common) direction of refinement.

In other words, dependently upon the situation, we effectively execute $h2$, $h4$, or $h8$ refinement of element $nel1$.

The algoritms are executed by calling routines $refineb$ and $refineb\_mod$, both from the $meshmodb$ directory. The routines can be called from the interactive mesh modification routine $meshmodb/mesh$ but only the modified version is avalaible from the graphical mesh modification routine $graph\_body/mesh\_mod$.

4.4 Data structure in Fortran 90 - continued.

During the h-refinements, the content of the data structure must be updated. First, the fathers and sons (i.e. the geneological information) of h-refined elements need to be specified. As new elements and nodes are created and/or deleted, their entire entries in the data structure must be created and/or deleted, and even some information for inactive elements needs to be updated. As the data structure for an initial mesh was described in

detail earlier, we will only list the changes necessary during h-refinement. Please refer to *modulidata_structure* for the definition of the user-defined *element* object.

Geneological information. After the interior of an element *nel* is refined, the element numbers of the corresponding sons: $nson1, nson2$ are placed in the entries $ELEMB(nel)\%sons$. Even though element *nel* becomes inactive after the refinement, its data structure information is maintained and updated, since the element information may be used to determine the constraints. Also, an unrefinement may activate this element again. Element *nel* becomes the father of elements $nson1$ and $nson2$, and thus the entries $ELEMB(nson1)\%father$ and $ELEMB(nson2)\%father$ contain element number *nel*.

Updating neighbors. For an initial, regular mesh, $ELEMB(nel)\%neig$ contains simply six 'equal size' neighbors of element *nel* across the six element faces. When elements are refined, the content of the $ELEMB(nel)\%neig$ array is modified and no longer can be interpreted simply as the active neighbors of element *nel*.

- If $ELEMB(nel)\%neig(i) = neln$ is positive then *neln* is the number of the *oldest neighbor* across the i-th side that has to be informed about possible changes on the side [8]. The point is that element *neln* may have been refined and its sons, grandsons, or even greatgrandsons may now be the active elements adjacent to the side. Most of the time, neighbor *neln* is either the *youngest unconstrained neighbor* of the side or the *oldest (possibly constrained) neighbor* of the side. Due to mesh modifications, however, this interpretation may not always be valid. For instance, due to an elimination of constraints, *neln* may no longer be the *youngest* unconstrained neighbor. Finally, the positive entry in $ELEMB(nel)\%neig(i)$ indicates always that element *nel* occupies the *whole* side and, in particular the corresponding mid-side node $ELEMB(nel)\%nodes(20+i)$ must never be constrained, i.e. the corresponding entry is positive and equal to the number of the mid-side node.

- If $ELEMB(nel)\%neig(i) = npos$ is negative then this indicates that element *nel* is adjacent to a 'big', i.e. unbroken side (i.e. its nodes and the corresponding d.o.f. are active). Flag $npos = 2, \ldots, 9$ indicates the relative position of the element side on the 'big side', as depicted in Fig. 6.

The modifications of the $ELEMB(nel)\%neig$ arrays are done *only in two routines: break_brick* and *break_side*, both from the *meshmodb* directory.

[8] Like node modification during p-refinements for instance

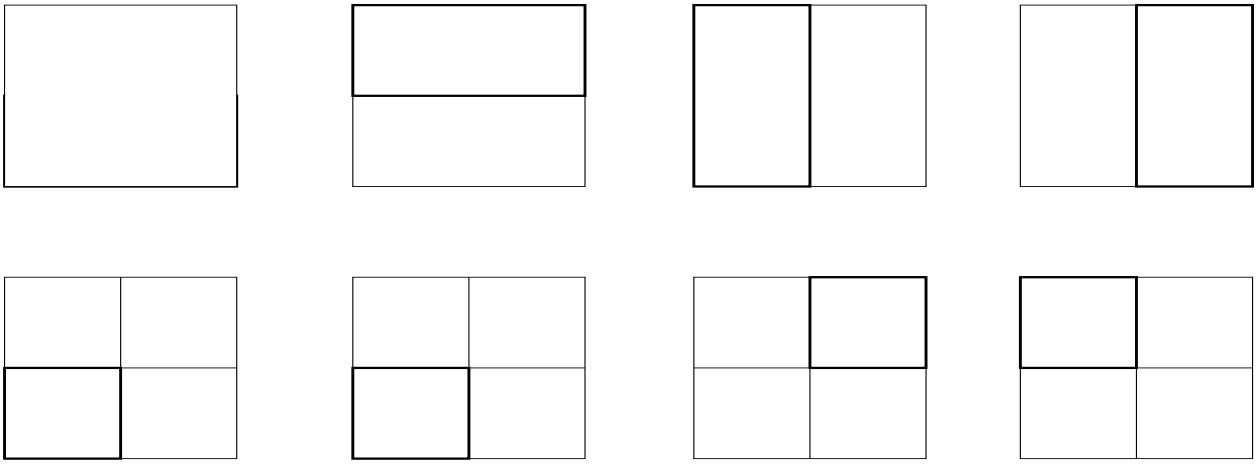

Figure 6: Classification of positions of a 'small element' side on a 'big' side

Determining neighbors. From the user point of view, once the initial mesh is h-refined, the information stored in the $ELEMB(nel)\%neig$ arrays is of no use. Instead, one has to use three routines: *neig_side*, *neig_edge*, and *neigbrb*, all from the *datstrb* directory. The routines return a complete information about neighbors of an element side, edge, or all neighbors across an element sides. The info includes not only actual, active neighbors for sides and edges, but also complete lists of all side or edge neighbors that have to be 'informed' about changing connectivities whenever the sides or edges are refined. The use of *neig_side* and *neig_edge* routines in all mesh modification algorithms is absolutely crucial.

The natural order of elements As in the 2D code, the binary tree data structure used in this code is used to define the so called *natural ordering of elements*. The ordering follows the numbering of elements in the initial mesh (done by the mesh generator) and the tree structure of elements following the *leaves* of the tree (*active* elements).

The algorithm is executed by routine *datstrb/nelconb*. A typical loop through all *active* elements in the mesh then looks as follows

```
nel = 0
do iel = 1,NRELEB
      call nelconb(nel, nel)
      ⋮
enddo
```

4.5 Constrained approximation for C^0 discretizations

We refer to [5] for a detailed discussion of the construction of global basis functions on irregular meshes, the resulted notion of the constrained approximation, the concept of the *modified element* and, finally, the definition of the three fundamental arrays allowing for the calculation of the modified element matrices:

$$Nrcon(k),\ k = 1, nrdofl$$
$$Nac(kp, k),\ kp = 1, Nrcon(k),\ k = 1, nrdofl$$
$$Constr(kp, k),\ kp = 1, Nrcon(k),\ k = 1, nrdofl$$

Here $nrdofl$ stands for the number of element local d.o.f., k is the index for a local d.o.f., $Nrcon(k)$ is the number of modified element d.o.f. *connected* to the local d.o.f. k, $Nac(:, k)$ lists the connected modified element d.o.f. and, $Constr(:, k)$ the corresponding constrained approximation coefficients. The essence of the constrained approximation package is to calculate these arrays for each constrained element, the unconstrained element is taken care in routine *logunbr* [9] discussed in the previous section.

The two main routines in the package are *logicmb* and *logicb*. The main task of the routine *logicb* is to eliminate possible double and triple constraints. For meshes with single constrained nodes only (e.g. those obtained using the *modified mesh refinements strategy*), routine *logicb* is inactive.

We shall proceed now with a direct discussion of the *logicmb* routine and its subsidiares, refering directly to the code.

4.6 The *logicmb* routine

Step 1: Reconstruction of info on constraints. The purpose of this step is to create two local data bases for nodes involved in the element approximation: one for regular (array nod_info) and one for constrained nodes (arrays $ncons$ and $list\_cons$). We begin by traveling the tree of refinements to identify the youngest unconstrained ancestor of the element. This may be its father, grandfather, or greatgrandfather. Let us discuss the most complicated case when the greatgrandfather is unconstrained. Subsidiary routine *logicb*1 uses the connectivity info stored for the grandfather and the greatgrandfather to reconstruct the info on constrained nodes of the grandfather in terms of their parent - greatgrandfather nodes. Constrained nodes are indicated by a negative entry in the $ELEMB(nel)\%nodes$. Once the

[9] If not otherwise explicitly stated, all routines discussed in this paragraph come from the *constrb* directory

info on the constrained node is reconstructed and stored in array *ncons*, the corresponding connectivity info on the grandfather node constrained node is updated to the (negative) number of the constraint. At the same time, the info on orientation of active nodes is stored in the local data base array nod_info. Next we proceed in a recursive way, by identifying constraints of father with respect to the grandfather. On input to the *logicb*1 routine we use now the updated connectivity info for the grandfather including the reference to already reconstructed constraints. This will allow for the future, purely algebraic elimination of multiply constrained nodes.

At the end of the first step, we have got the two data bases, one for the unconstrained nodes and one for the constrained ones, and an updated info on connectivities of original element Nel including the constrained nodes. The info on constrained nodes is done in the recursive way, i.e. parent nodes of constrained nodes may be constrained themselves.

We emphasize that this step is done *exclusively* at the nodal level. This isolation of the operations on nodes from those on d.o.f. is essential in customizing the code for $H(curl)$ - approximations necessary for electromagnetics.

Step 2: Establishing lists of the modified element nodes. By the definition, the nodes of the modified element corresponding to an element *nel* include active nodes of element *nel* and the parent nodes of its constrained nodes. In the case of multiple constrained nodes, the grandparent or even greatgrandparent nodes have to be included. Vertex nodes are separated from higher order nodes and both lists for the modified element are determined in the subsidiary routine *logicb*2. Two facts are important:

- In general, the lists include most but *not all* nodes stored in the local data bases.
- At this point, the lists of nodes for the modified element *do include* constrained nodes and all constraints are view as single constraints, except that some of the parent nodes are themselves constrained nodes. This makes determination of the corresponding constrained approximation coefficients easier.

Following the lists of the modified element nodes, each corresponding d.o.f. is assigned the corresponding number on the list of the modified element d.o.f. For vertex nodes, this number coincides with the location on the list (each vertex node has only one d.o.f), for higher order nodes, it is stored in array *na*2. Hence we use the name *na*2-function.

Step 3: Evaluating the constraints approximation coefficients. In the final step we loop through all the nodes of element *nel*. If the node is unconstrained, we determine the

corresponding entries in the $Nrcon, Nac, Constr$ arrays the same way as in routine *logunbr*, otherwise we call the third subsidiary routine *logicb*3 which determines the corresponding constrained approximation coefficients. This is perhaps the most technical operation here, related closely to the collected info on the orientation of unconstrained nodes and a precise classification of constraints. We encourage the reader to compare carefully the info in routine *logicb*1 with the corresponding evaluation of the constrained approximation coefficients in routine *logicb*3.

Finally, at the end of this step, the constrained approximation coeffcients are also determined for all parent nodes that have been listed as nodes of the modified element.

4.7 The *logicb* routine

Besides deciding whether to call the unconstrained approximation routine *logunbr* or the constrained approximation routine *logicmb*, the main task of routine *logicb* is to eliminate possible multiple constraints. The elimination is done in a recursive way. We first consider constrained nodes of the father with respect to the grandfather, and then the constrained nodes of element Nel with respect to its father. This order of eliminating multiple constraints is essential. For each of the d.o.f. of a constrained node, once we encounter a constrained parent node, we add the d.o.f. corresponding to the grandparent nodes to the list of connected d.o.f. and recalculate the corresponding constrained approximation coefficients. The procedure is identical with the one discussed in [5] for 2D meshes.

4.8 Additional comments

Assembling of global matrices. Interfacing with solvers. The global assembling procedure is identical with that for regular meshes discussed in the previous section. Once the modified elements matrices are calculated, the regular and irregular meshes are treated in an identical way. This is especially convenient when it comes to writing an interface with a linear equations solver. In particular, the interface with the frontal solver discussed in the last section remains exactly the same. In principle, the constrained approximation has been entirely hidden from the user in the constrained approximation routines. The user is free to modify the element routine (in particular to accomodate different boundary-value problems) and the constrained approximation routines will automatically produce the modified element arrays allowing then again for a standard interface with a solver.

Evaluation of local d.o.f. Routines *constrb/nodcorb* and *constrb/solelmb* return for an element *local* geometry and actual d.o.f. necessary to evaluate the geometry or the solution (e.g. for postprocessing). The evaluation is based again on the constrained coefficients stored in arrays $NRCON, NAC$ and $CONSTR$. First, the corresponding modified element d.o.f. are copied from the data structure arrays into a local vector $dofmod$ and then the local d.o.f. vector dof is evaluated using the algorithm:

initiate dof with zeros
for each local d.o.f. k
    for each connected modified element d.o.f. $j = 1, \ldots, NRCON(k)$
        $i = NAC(j, k)$
        accumulate for the local d.o.f.:
        $dof(k) = dof(k) + CONSTR(j, k) * dofmod(i)$
    end of loop through connected d.o.f.
end of loop through the element d.o.f.

In this way, the conversion of the global d.o.f. to the local ones is again hidden from the user who needs only know how to call the two routines.

5 Organization of the Code

The code is organized in the following subdirectories:

- *commons* - system common blocks,
- *constr_util* - constrained approximation utilities,
- *constrb* - constrained approximation routines,
- *datstrb* - data structure routines,
- *elem_util* - element utilities,
- *ffld* - free field reader and debugger routines,
- *files* - system files, sample input files,
- *frontsol* - frontal solver routines,
- *gcommons* - graphics common blocks,
- *geometry* - Geometrical Modeling Package routines,
- *graph_body* - actual graphics routines for the code,
- *graph_geom* - graphics routines for the Geometrical Modeling Package,
- *graph_interf* - graphics interface routines,
- *graph_util* - graphics utilities,
- *hp_interp* - *hp*-interpolation utilities,
- *laplace* - element routines for the Laplace equation model problem,
- *l2proj* - element routines for the L^2-projection model problem,
- *lin_algebra* - linear algebra routines,
- *main* - main program, driver for the code,
- *meshgen3* - initial mesh generation routines,
- *meshmodb* - mesh modification routines,

- *module* - data structure moduli,
- *solver1* - interface with frontal solver,
- *utilities* - general utilities.

Most of the essential packages have already been commented on in the previous chapters. All routines in the code are fully commented and (most of the time) follow the coding protocol. We shall continue now with a brief description of selected directories which have not been dicussed yet.

5.1 Free field reader

A standard Fortran code reports an error and stops if a data read from a file is inconsistent with the declared data type in the program. The free field reader routines allow to read in the data without the danger of stopping the execution of the code. This is especially useful during interactive calls to graphics routines.

The reader has been designed to input data in a standard format only. The format consists of a sequence of characters separated by comas. In between the comas, a typical data consists of a *name*, the equality sign =, and another string of alphanumeric characters or a number. The equality sign and the last string are optional. Here is a simple example:

```
nrbreaks = 5, lprname = test, level = 3, twodim
```

The whole sequence has to be terminated with the return key.

For the user's convenience, the graphics routines are provided in two versions, with and without the use of the free field reader. Please consult the two versions of routines *graphb* and *grbody* to learn how to use the *ffld* routines. Besides the graphics routines, the free field reader is used only in the debugger described below. Thus, if the user chooses not to use the debugger, he/she may also skip using the *ffld* routines as well.

5.2 The debugger

The code comes with a very simple, platform independent, internal debugger. Its use consists just of introducing in the code control prints that can be controlled interactively when executing the code. This should not be confused with local printing flags that have been placed in most of the routines and that can be activated only by recompiling the routines. The basic features of the debugger are as follows.

- Control prints are identified with user specified names that usually coincide with names of the routines in which they have been placed.
- There may be an arbitrary number of control prints introduced in the code but only the ones that are listed in the *debug* file *may* be activated. The actual activation of the control prints from the list is done *interactively*, any of them can be activated or deactivated at any time after the execution stops at a control print.
- If the list of parameters for the routine being debugged includes element number Nel, the corresponding control prints may be activated for a specific element number only. This is very convenient if an error occures only in a specific element.
- The control prints can be introduced into every routine in the code. Their format has been standarized which helps the user to learn after a while to ignore them when reading the code.

Example

- File *files/debug*:

```
nrbreaks = 5
lprname = test
lprname = xxxx
lprname = test1
lprname = timestep
lprname = profile
@
```

  The sign @, placed at the end, tells the free field reader to switch reading from file *debug* to the terminal (keyboard).

- The routine with debug prints:

```
subroutine test(Nel,...)
dimension array(3,100)
character* lprname
common /cdbg/ isolin21,lsolin22,...,lsolout,lprints(20)
common /cdbg1/ nrbreaks, lprname(20)
save irout,idbg
if (irout.ne.100) call initdbg('test',irout,idbg)
```

```
⋮
        if ((lprints(idbg).eq.1).or.(lprints(idbg).eq.(-Nel))) then
          call dbgprint(1,'test','Nel',Nel,rdummy,1,1,1,1,11,0)
          call dbgprint(2,'test','array',ndummy,3,1,2,1,10,1)
        endif
⋮
c
c... Analogous calls to dbgprint in as many places as you wish
⋮
        end
```

See routine *dbgprint* for the explanation of the control print parameters.

- Debugger commands. The following commands are available when the program stops and executes the control print.

g	proceed,
quit	stop the program,
test=1	activate the debug print named *test*,
test=-57	activate debug print *test* for element 57 only,
test=0	disactivate *test* debug print,
status	display status of all debug prints.

REMARK 2

1. The debugger is initiated in the main program by reading the number and the list of the debug prints to be (possibly) activated.

2. Consult files *files/debug* and *files/debug.save*. If the number of control prints in *debug* is set to zero, the debugger will not be activated. Once the number, however, is different from zero, and the debugging flags have been specified, the debugger is activated and the execution is paused. At this point, even if we decide to proceed *without* debugging and type the **g**(o) option, we have to "cheat" the debugger first with *a* statement that sets some of the control prints to a positive integer greater than 1, e.g. `solout=10`. The statement itself need not to make any sense. Typing **g** next will execute the code in the usual way.

∎

5.3 Graphics interface

In order to run the code, besides a FORTRAN 90 compiler the user will need an interface with the computer graphics. The C routines from directory *graph_interf* provide such an interface with standard X-Windows graphics. The interface allows additionally for producing a PS copy of any picture that appears on the computer screen.

5.4 Graphics packages

The graphics routines have been organized in three directories. Directory *graph_util* contains simple utility routines and routines that are common to all graphics programs. It is assumed that each image is generated in the form of a collection of triangles that may overlap each other. This allows, in particular, to achieve the hidden line effect by listing the triangles in the reverse order according to their distance from the observer. Each graphics program generates the data for the triangles (geometry, color, border lines, symbols to be written out etc.) and stores them in arrays listed in graphics module *module/graphmod*, see the information therein. Routine *graph_util/dpvisid* decodes the information about the triangles to be drawn (geometry, color, border lines, symbols to be written out etc.), and displays them on the computer screen.

Directory *graph_geom* contains graphics routines for the *Geometric Modeling Package* [2]. They allow to display the geometrical object being modeled and help to debug the input file for GMP.

Directory *graph_body* contains graphics routines for the actual 3D *hp*-meshes visualization. The two main routines are *grbody* and *meshb_mod*. Routine *grbody* allows to display a current *hp*-mesh with a color code illustrating the distribution of different orders of approximation. The routine also displays a contour map corresponding to the approximate or exact solution. Finally, routine *meshb_mod* allows for an interactive mesh modification.

Warning!! Numbers of elements to be refined are reconstructed from coordinates of points on the screen input by clicking the mouse. The reconstruction, done in routines *elem_util/elnumb, elem_util/invmapb_side*, is based on the solution of a nonlinear system of algebraic equations relating master element coordinates with physical and screen coordinates. The algorithm *may not converge* if the indicated element side is only *partially* visible on the screen. This happens frequently in the case of large, curvilinear elements that may "wrapp" around themselves. In such a case, one should change first the point of view to make sure that the whole element side appears on the screen.

5.5 Sample input files

A few sample input files are provided in directory *files*. The input includes data for *GMP* and the *hp* mesh generator. We refer to [2] for a precise description of the input format for the *GMP* package. The input for the mesh generator includes the following lines.

1 Number of geometrical blocks (hexahedra) constituting the object.

2 For each block:

- block nickname, as used in *GMP*,
- order of approximation and number of subdivisions for the block, in directions of axes x, y, z (total of six integers),
- a six digit nickname encoding boundary conditions flags for the block.

3 Workspace allocated dynamically for the frontal solver.

4 Maximum number of nodes, elements, equations, and the actual numebr of equations being solved (just one for the model problems discussed), a total of four integers.

References

[1] L. Demkowicz, L. J.T. Oden, and W. Rachowicz, "Toward a Universal *hp* Adaptive Finite Element Strategy, Part 1. Constrained Approximation and Data Structure", *Computer Methods in Applied Mechanics and Engineering*, **77**, 79-112, 1989.

[2] L. Demkowicz, A. Bajer, and K. Banas, "Geometrical Modeling Package", *TICOM REPORT 92-06*, August 1992, The University of Texas at Austin, Austin TX 78712.

[3] L. Demkowicz, and J.T. Oden, "Application of hp-Adaptive BE/FE Methods to Elastic Scattering", in a special issue on p and hp methods, edited by I.Babuska and J.T. Oden, Computer Methods in Applied Mechanics and Engineering, 133, 3-4, 287-318, 1996.

[4] L. Demkowicz, K. Gerdes, C. Schwab, A. Bajer, and T. Walsh, "HP90: A General and Flexible Fortran 90 *hp*-FE Code", *Computing and Visualization in Science*, **1**, 145-163, 1998.

[5] L. Demkowicz, T. Walsh, K. Gerdes, and A. Bajer, "2D *hp*-Adaptive Finite Element Package. Fortran 90 Implementation", *TICAM Report 98-14*, July 1998.

[6] W. Rachowicz and L. Demkowicz, "A Two-Dimensional hp-Adaptive Finite Element Package for Electromagnetics", *TICAM Report 98-15*, July 1998, accepted to *Computer Methods in Applied Mechanics and Engineering.*